DOCUMENT RESUME

ED 338 227 IR 015 263

AUTHOR Booth, Shirley
TITLE Conceptions of Programming: A Study into Learning To Program.
INSTITUTION Goteborg Univ., Molndal (Sweden). Inst. of Education.
PUB DATE 90
NOTE 30p.
PUB TYPE Reports - Research/Technical (143)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Computer Science Education; *Concept Formation; Foreign Countries; Heuristics; Higher Education; Interviews; *Learning Processes; Microcomputers; Problem Solving; *Programing; *Programing Languages; *Undergraduate Students
IDENTIFIERS Sweden

ABSTRACT

This paper reports the results of a phenomenographic study which focused on identifying and describing the conceptions of programming and related phenomena of about 120 computer science and computer engineering students learning to program. The report begins by tracing developments in the students' conceptions of programming and its parts, and then looks at the development of their understanding of programming concepts and techniques. A group of undergraduate students at the University of Gothenburg (Sweden) were followed during their 6 months of study, during which time they received instruction in two programming languages--SML (Standard Meta-Language) and Pascal. The students were interviewed six times during the study, and the results of the study are based on thorough analyses of the first and sixth interviews, plus interviews with a group of 12 second year students who had completed a further course of programming. Data from "Programmers at Work" (Lammers, 1986), a collection of interviews with successful professional programmers, were used to broaden the base of the study. Analyses identified students' conceptions of programming as an activity, a problem to be solved, and an end product; computers as tools, facilitators, simple machines, and universal engine conceptions; and programming languages as codes, utilities, a means of communication, and a means of expression. The report concludes with discussions of complexity in the conceptions of programming, learning to program, and further research to be conducted as a continuation of this study. Descriptions of the programming languages and an outline of the interviews are appended. (11 references) (DB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☐ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

ED338227

Conceptions of Programming:
A study into learning to program

Shirley Booth

Report 1990:01
Institutionen för pedagogik
Göteborgs universitet

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

Shirley Booth

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

120f263

BEST COPY AVAILABLE

Conceptions of Programming:

A study into learning to program

Shirley Booth

December 1989

Abstract

The paper describes a phenomenographic study of a number of university students learning to program. The students, first and second year students of computer science and engineering, learned programming in two different sorts of language, the first a very high-level functional language called Standard Meta-Language (SML) and the second a high-level procedural, imperative language, called Pascal. They were interviewed a number of times and transcripts of three of these sets of interviews have been analysed.

The results presented concern conceptions of programming and two major components of programming as an activity, namely the computer and programming languages. Three conceptions of programming are identified, namely programming as an activity oriented towards the computer, the problem and the product respectively. Four conceptions of the computer are identified: the tool, the facilitator, the machine and the universal engine conceptions. Four conceptions of programming languages are found: programming language as a code, as a utility, as a means of communication and as a means of expression. These terms are elaborated and justified with extracts from interviews.

The results are discussed in terms of their complexity; learning to program can now be seen as the learner developing more advanced conceptions of these components of programming, but at the same time retaining the less advanced conceptions for appropriate usage, and developing more complex relationships between the components. Two cases are studied as examples.

Introduction

Computers have been with us for forty years but the growth in their use has been exponential. In 1950 there were a small number of machines, mostly being developed, and primarily for military purposes, in the US and UK at least. By 1960 they had become rather more common, computers being used for administrative purposes in government and big business as well as scientific research and military uses. They were still exclusively in the hands of an elite for their use demanded insight into arcane codes and coding methods, and they required housing in vast air-conditioned halls. By 1970 all businesses of any size had a computer of their own, or bought time on one, to administer their affairs. They were going through a phase of decentralization, research departments for example having their own, often special purpose, machine instead of sharing one relatively big central machine. But by 1980 computers had shrunk in size and cost while increasing in power, and small machines that could produce arcade games on the home TV were every 13-year olds dream Christmas present. Now 1990 is here and there are very few people who do not interface directly with a computer in some way - they are everywhere, in the workplace, the home and in schools.

All this can be explained in technical terms - silicon chip technology and generations of hardware - or social terms - the technological revolution and an implicit belief in endless progress measured in terms of advances in gadgetry - or in the terms of improved efficiency and management used in business. The fact remains that computers are here, they are everywhere around us, everyone is affected by them, almost everyone uses them in some sense. While the great majority are not aware of exactly how they are making use of a computer, a sizable and growing number of workers are involved in making computers do their bidding. There are those for whom the computer is a tool of their work, for text processing, or administration, or calculation, or control. There are others who make the computers do those tasks, the computer engineers who devise the equipment itself and the software engineers who handle the programming of the equipment.

What is meant by programming in the last sentence? The unique thing about the computer as a machine is that it provides its user with not one tool but many. Its function at any one time is determined by the instructions it has received and holds and the way in which it can execute these instructions. Such sets of instructions are determined and put together into a program by a programmer, or software engineer. That sounds very simple, but in fact it is almost always an extremely complex and intellectually demanding process. Whether programming is a skill or a science or even an art is open to debate; there are features of all three. Certainly, programmers at work are exercising a skill, but a mathematical science lies behind it, even if it is sometimes seen as underpinning existing skills and offering cures for some bad habits rather than as providing an essential foundation for the skills.

This paper will report the first set of results from a project to investigate learning to program. This particular study was motivated by the need to study learning to program by a particular group of people, namely university students of computer engineering, who will be a part of the next generation of computer and software developers. In general, it is such students who need to develop the most advanced skills during their studies, and who also need to acquire the soundest foundations in the science of programming. The study considers programming as a complex activity in which the programmer involves the computer, programming languages and his or her programming skills to analyse and solve some problem in the real world, either for himself or herself, or for others. It is an attempt to trace first of all the developments in conceptions of programming and its parts, and then in the light of this look at the development in understanding programming constructs and techniques. This paper will present the conceptions of programming, the computer and programming languages found from the study, and on the basis of these suggest a structured way of describing, at the non-technical level, learning to program.

There have been other studies of learning to program, but not with this particular interest. For example, Andersson has studied well-educated people learning to program with the explicit aim of fitting their experience into the structures and rules that go to make up his theory of cognition, ACT* (Andersson, 1984). The research interest is in general theories of learning within a cognitive science framework. Soloway and his colleagues have studied both school and university students learning to program, mostly with special emphasis on the errors they make when writing particular pieces of program code (Soloway, 1985). In this case the aim is to build such knowledge into a computer-based tutorial system. Linn and her colleagues have studied school students learning to program, one interest being in the hypothesis that learning to write computer programs is a problem-solving skill that is transferable to problem-solving in other areas of study (Linn, 1985). Mayer and colleagues have looked at students learning the first elements of programming, but generally with an interest in the differences induced by offering different learning aids and their relevance for instructional strategies (Mayer, 1981). Here programming simply serves as a field in which it can be assumed that the students are all starting from the same zero level of competence.

These studies and others are discussed more thoroughly elsewhere (Booth, 1986) but are mentioned here to indicate the sort of research in the field. While learning to program is in focus, the aims are rather diverse, and even if they all have an interest in improving instruction in some way, that aim is secondary to the more general interests of psychology and cognitive science in particular. The study to be presented here aims to maintain a more holistic view of programming.

Method

This study has its base in another tradition of research into learning, that of phenomenography. Phenomenography offers a way of describing learning in terms of developing conceptions and is described fully in other reports (Marton, 1986). Suffice it to say here that by making careful interviews with students about the content of their studies, it is observed to be possible to identify distinctly different conceptions of the essential phenomena in that content. The methodology is well developed and its use has been described in a number of subject areas, such as arithmetic skills among small children (Neuman, 1987), the nature of matter (Renström, 1988) and Newtonian mechanics (Johansson, Marton & Svensson, 1985). If certain conceptions are considered to be qualitatively more advanced than others, then moving towards them can be considered to be learning. Thus, to give a very simple example, if a student who initially conceived the application of a force to be necessary for maintaining a body's uniform motion later perceives that the force will actually result in an acceleration, then irrespective of his ability to apply the formulae and equations associated with Newton's laws, he can be said to have learned the concepts. It is a well-documented fact that students can very often solve the problems that are designed to test their learning without understanding the concepts, so this approach to studying learning offers a powerful alternative.

A group of computer science and computer engineering undergraduates at the Chalmers University of Technology, in Gothenburg, Sweden, were followed during their first six months of study, during which time they received two courses of instruction in programming. The first course involved programming in a very high-level functional programming language called SML (Standard Meta-Language) and the second half took up programming in a high-level imperative procedural language, Pascal. The essential elements of the languages are described in the Appendix A, but let it be said here that programming in SML lies close to specifying the problem to be treated in terms of the data structures and algorithms available within the language, whereas programming in Pascal involves detailing

those data structures and algorithms. The more or less explicit aim of arranging the course in this way is to teach the student to specify a problem before detailing a solution.

Before the course actually started all the students (about 120) were given a questionnaire asking for details of their educational and computing backgrounds. On the basis of these, 15 students who were also willing to take part in the study were selected as being representative in terms of experience. One changed to studying physics almost at once but the remaining 14 lasted the course as far as the interviews went, even though one actually stopped studying and returned to work. They were interviewed six times in all, once before the course started, once after it ended and the rest at roughly equal intervals. The interviews and their transcripts form the basic material for the study of learning programming. In addition, a group of 12 second year students were interviewed after they finished a further course of programming, in order to broaden the base of the material for the purpose of identifying conceptions of the central phenomena. In the report that follows all the first year students have been given men's names, in spite of the fact that two were female, and the second year students have women's names, three actually being female. This is to avoid identification of individuals, who were guaranteed anonymity.

The interviews have three aspects which aim to cover the phenomenon of programming in a global sense. First, they all take up some of what will be called the non-technical aspects of programming (even though they are rather technical in fact). By this is meant such topics as the nature of programming, the nature of computers, programming languages, and studying and learning programming. Secondly, certain interviews take up specific technical aspects of programming, in particular the constructs of the the two programming languages involved. Thirdly, there is a problem-solving session in some interviews, when the students either tackle a new problem or describe how they approached and worked at exam, lab or exercise problems. In Appendix B there is a summary of the topics discussed and the area which each was intended to illuminate.

This report focuses on identifying and describing conceptions of programming and related phenomena, and is based on a thorough analysis of the two of the six interviews, namely the first and the sixth, which treat such topics most extensively, plus the interviews with second-year students. All three have been used for analysis of the phenomena of programming and programming languages but only the first first-year and the second-year interviews were appropriate for considering conceptions of the computer.

A further set of data which has been of value in providing another perspective is that contained in a book called Programmers at Work (Lammers, 1986). This is simply a number of interviews with successful professional programmers in which they are asked about all aspects of their work - not only software but also hardware and entrepreneurship. While this cannot contribute to the aim of studying learning to program, it can, like the second-year interviews, broaden the base of the study. When cited, the professionals are identified by their initials.

Results

Conceptions of programming

The phenomenon which is central to this study is that of programming, programming in the global sense, divorced from the language used and problem to be tackled. Thus it is fitting to take it up first, even though because of its complexity it will have to be returned to in the Discussion section. First there is a short description of the three distinctly different conceptions which have been identified, and then each will be taken up and elaborated.

The three conceptions of programming are most clearly distinguished by their prime focuses, or orientations, and are:

1. **Programming as a computer oriented activity**, in which programming is conceived of as an activity that focuses on the computer
2. **Programming as a problem oriented activity**, in which the main focus is on the problem that the programming activity is intended to solve, rather than on the computer itself, and
3. **Programming as a product oriented activity**, in which the main focus is on the program as a product, in the sense that programming is an activity for producing programs that will be bought and used, and/or accessed for maintenance by other programmers.

Computer Orientation

The single distinguishing feature of statements that imply the computer orientation conception of programming is that the role of the computer is stressed. One such statement is from Greg in the first interview:

> I. What is programming actually?
>
> G. Well, you sit at the computer and play a bit at the keyboard (*Knappa lite*)

Another is from Eric:

> E. It's about making the computer usable. [..] You have to write a program in order to use the computer for something

another from David:

> D. You have to have a program before the computer will do anything at all

and a fourth from Clive:

> C. It's to sit in front of the computer screen and .. ah .. hm .. how can I put it ... to get the machine to achieve something, something that you want it to do

All these, all of which come from the first set of interviews, show a clear conception of the essence of programming involving the programmer in a one-way relationship with the computer, making the computer do something (unspecified), in some way (also unspecified).

A development of this one-way relationship in an unspecified activity is expressed by John:

> J. It's composing, or putting together, a plan of action for the computer which is to do something in a certain way (...) It's putting together instructions for the computer.

in which the activity starts to take form, that of putting together a plan of action for the computer to follow.

Here is a quote from the book Programmers at Work, in which JS reminisces on an early job:

> J. [..] It was my first programming job where I was able to sit down and actually have a conversation with the computer.

This too implies a computer oriented view of programming (even if only in retrospect) with the significant difference that there is a two-way relationship. The programmer and the computer hold a conversation.

Problem orientation

The problem orientation is implied by statements in which the role of the computer is not mentioned, or is insignificant. What is in focus is the problem that gives rise to the need to program. Adam makes three separate statements in his first interview which indicate this conception:

> A. You make a program
>
> A. You have a task to solve and you try to make a program of it.
>
> A. .... I suppose you have to start from having a problem that has to be solved, and first it needs a hell of a lot of thinking, I suppose ... wondering about how to make it good, and then ... err ... well it is the typing itself I suppose.

He develops a problem orientation here from a totally non-specific conception, (although jokingly). He mentions typing, which alludes to the computer conception, but it is insignificant compared with the focus on the problem, thinking about the problem, and solving the task. In his first interview Brian describes programming thus:

> B. It's about getting a problem on paper and having to solve the problem, or teach the computer to solve the problem, so that you don't have to sit there and solve it. Yes, try to teach it as easily as possible, so that it can do it as well as possible, sort of ...

Here again, while the computer is mentioned it is secondary in importance to the problem.

Mark responds to a question "What is programming actually?" in this way:

> M. Well, basically, it is the typing in itself, when you tell the computer what to do. Uh, actually, programming is designing some program or other, the whole thing, which you do before you even get near the computer. [Long pause] Mm. Programming - what shall I say - basically it's having a problem that you want the computer to solve, and first you have to think out how to tackle the problem - now for me that isn't programming but it's preparation for programming. Then when I know how I'm going to tackle the problem, that's when the programming starts so to speak, you have to sort of think out how the program is going to look, and you put it into the computer. It's sort of the same stage I think. I don't know, but I think so.

This is interesting in that it both starts and ends in an expression of the computer oriented conception, but in the middle, while denying that it is programming, he takes up the area of designing the program to solve a problem. It is appropriate to take up here a possible explanation for this uncertainty, and it lies in the typical job specifications in the computer business. Traditionally, in a computer department, whether involved in administrative or scientific applications, there have been two kinds of worker — the designer and the programmer. The designer researches the problem area and comes up with a solution, specified precisely in terms of the required program rather than in terms of the problem. Then the programmer comes along and writes code in an appropriate programming language, puts the program together, runs preliminary tests and hands it back to the designer for evaluation. This is a division in the programming process created because of the different skills needed —the designer in an administrative department might be a management expert while the programmer has been trained to code according to a given specification. However, in the context of computer engineering education this division is irrelevant because the computer engineer, if he programs at all, will be responsible for both

aspects of the task. Furthermore, there is a move away from this division, brought about by a flatter structure in the industry and the availability of more powerful programming tools. So, when a student talks in these terms — and it happens a couple of times in the first interview — it may be interpreted as though these job descriptions met during periods of work experience are confusing the issue, and for the purposes of the study both parts can be taken under the umbrella of programming. Thus Mark here expresses both the computer orientation and the problem orientation, and his denial that the latter refers to programming can be put aside.

Product orientation

The third conception of programming focuses on the program that is produced rather than on the problem which gave rise to it or the computer's role. There are two distinct focuses; first there is the focus on the program as an object that will be used by other programmers, in maintaining it in the future. Keith expresses this clearly in the first interview:

> K. He should write programs that everybody can understand, even if he's dead and buried, you should be able to understand his programs later {...} Those who use them might have changed their requirements after a few years, you might want to make small changes. And then if nobody knows where they should be changed, you have to get new programmers who have to do everything over from the start again

The second focus is on the program as something that users will use, and that the program should be easy for them to use, as expressed by Brian:

> B. They make good programs that work well, so that the user understands, doesn't understand what is happening maybe, because that's ... but understands what he has to do sort of. I think there are a lot of programs with poor information coming up on the screen sort of, they should be easy to follow.

It might be thought that these two distinct cases of the product orientation should be separated, but in fact they are often expressed at the same time, and sometimes the product orientation is expressed without one or other of the focuses, but rather expressing the common concerns. For example, Philip refers to the need for thorough testing

> P. It [the program] should be usable by those for whom it is intended, and tested many times as well, it should be tested a lot even for extreme cases I think. That's important before you release it.

and others refer to the need for good documentation. Testing and documentation are both connected with the final product, and are in the interests of both the user and the subsequent maintaining programmer.

Summary

This section of the report can be ended with a few notes. First, all the quotes from interviews are taken from the first interview, with the exception of JS's from Programmers at Work. This is not because they occur only there, but rather because they appear there in a much less complex form than in later interviews. The complexity will be discussed in a later section, after taking up conceptions of some other central phenomena which go to make up the complexity. Secondly, there is a dimension to the conceptions of programming which has not yet been mentioned here, and which none of the quotes yet illustrate, and that is the time-dimension. The less experienced, and that is virtually all the first year students throughout the period of the study, discuss programming as though it consists of a single burst of activity, while the more experienced, and the professionals in particular, discuss it as a long and often drawn out process. It is another aspect of the complexity of the phenomenon of programming and will be returned to later.

Conceptions of the Computer

Four distinct conceptions of the computer have been identified and will be described here, first briefly then more thoroughly. The four conceptions are:

1. **The computer as a tool**, in which the computer is seen as being equipped to perform certain tasks — one can use the computer to do lots of things that would otherwise be more difficult, such as calculations, or text-processing.
2. **The computer as a facilitator**, the computer has the potential to *enable* one to do the things one wants to do, they facilitate actions or wishes, they offer the potential to fulfil needs.
3. **The computer as a machine**, the computer can perform a number of specific tasks, be made to act like a number of specific machines, and
4. **The computer as a universal machine or "universal engine"**, in which the computer is seen as having no limit to its possibilities — given enough ingenuity one can make the computer do absolutely anything.

The tool and the facilitator conceptions

The four conceptions listed above clearly form two pairs. The tool and the facilitator conceptions both focus on the tasks performed by the computer; the former emphasizes the sorts of applications that are made while the latter emphasizes the potential for application. The following two quotes, both taken from the first first-year interview, illustrate the difference. Keith gives expression to the tool conception when he says:

I. What can computers be used for?

K. Calculations, administrative tasks, control of different processes, at Volvo for example

I. On the line?

K. Yes, on the line, robots have already taken over there, adjustable robots

whereas Brian shows the facilitator conception by saying:

B. [...] I think the computer is going to take a bigger place in society, that's clear. We are going to get more help from them, a lot of heavy jobs will possibly disappear [...] people won't have to sit and sort through a mass of paper because it can be done in the computer instead ...

Both of these refer to what the computer can do for its users, what use can be made of them. One way of describing the difference between them is that while the actual processes performed by the computer might be identical, the subject's view of them is different - in the tool conception the relationship between the user and the computer is taken for granted and seen as fixed whereas in the facilitator conception a more active role is given to the user/society in the relationship.

The machine and universal engine conceptions

The second pair, the machine and universal machine conceptions, concern the perceived limits of the uses to which the computer can be applied. The machine conception of the computer is quite simply that the computer is no more or less than a machine, special only in as far as it can take on the role of many different machines. This is implicit in a great deal that is said in direct reference to the computer - computers do things, computers have to be instructed or controlled - and is implied by statements to the effect that computers have no feelings, and that computers can only do what their programmers make them do. Mark expresses both parts of this:

M. [...] Last year I worked in the chemical industry and there there are computers for control and absolutely everything

I. Where might the limits lie?

M. The limits are actually what you yourself believe the computer can do, sort of ... I mean, the computer won't do anything it's not told to do, will it? So I think that the computer has no limits, but it is man himself that sets the limits.

This latter point also hints at the second in the pair, the universal engine conception, by which is meant that the computer can do absolutely anything at all. It is striking how many people in the first set of interviews said that computers can be used for doing everything. While Mark above says that the computer has no limits but it is man that sets the limits he deliberately steps back from the universal engine conception of the computer (while not explicitly denying it). The full-blown conception involves belief that the computer can indeed do anything, it is merely a question of time and effort and ingenuity. Eric says in the first interview:

I. What can computers be used for?

E. Everything!

I. Everything?

E. Yes ... control, process control and that sort of thing ..

He goes on, when prompted, to list lots of things computers are involved in and might be used for, and finally:

I. Is there anything that computers cannot be used for?

E. Yes of course there are ...Feelings and all that, the computer cannot show feelings

The universal engine conception is referred to explicitly in Programmers at Work by BL:

Q. Why do you think people are so fascinated by the idea of AI?

A: Well, I'm not sure. Part of it is the basic computer fallacy that states that the computer is a universal engine that can do anything. If there is no obvious way to show that something is impossible, then some people assume it must be possible. A lot of people don't understand what the consequences of complexity are, and without that understanding they are likely to get burned. (BL, PAW p29)

BL is not expressing the conception himself but is referring to it being held by others.

The most striking distinction between these two conceptions is that the machine conception implies, or claims explicitly, that there are a limited number of functions that the computer can be expected to take, while the universal engine conception implies, or again states explicitly, that there is no such limit.

Summary

Four conceptions of the computer have been identified and described here — the tool, the facilitator, the machine and the universal engine conceptions — which can be seen as two pairs. The tool and the facilitator conceptions both refer to what the computer does for its user but they focus on different relationships between the computer and the user. The machine and the universal engine conceptions both refer to the functions that the computer can fulfill, and their limits, or lack of limit.

Conceptions of Programming Languages

Discussion of programming languages take place throughout the interviews, either deliberately brought up as a topic for exploration, or incidentally in the context of other topics. The results to be presented now come from the first and sixth interviews with the first year students and the interviews with the second year students, in which the topic was specifically taken up, using lead questions such as "Why do you think there are so many programming languages?" and "What do you think goes to make a good programming language?" as well as incidental discussion about the two languages that had been studied.

Four conceptions have been identified, namely:

1. **Programming languages as a code**, in which the language is seen as a set of instructions, commands, symbols and constructs;
2. **Programming languages as a utility program**, in which the language is seen as enabling the programmer to achieve certain effects;
3. **Programming languages as a means of communication** between the programmer and the computer, in which the language is the means the programmer has to make the computer do the tasks required; and
4. **Programming languages as a medium of expression**, in which the programmer can express the solution to a problem or idea in such a way that the computer can effect it.

Now these four can be exemplified and refined with reference to the three sets of interviews named.

Programming language as a code

In using the word code to name this conception, it is not meant in the sense of a one-to-one correspondence, such as is used to send secret messages. What it is intended to convey is more the sense of a formal set of symbols and rules; the word "formalism" could convey this sense, but then at a more abstract level than is required. In linguistic terms, as used within computer science, it refers above all to the syntax and semantics of the language. Students refer to the intuitive nature of a programming language — that a word used for a command, such as PRINT, should correspond in meaning to its meaning in natural language, and that + should mean add rather than multiply. They refer often to the structure of the language, and when this means the way in which the program shifts control from one part to another, it can be taken to indicate the code conception.

Adam, who has virtually no experience of programming prior to the course, expresses the idea of the programming language being a collection of words and symbols in the following extract, taken from his first interview:

I. How would you judge whether a language was good or not?

A. [..] I would like to compare it with a camera, one of these new cameras that are so electronic and everything. ... you can treat them entirely intuitively. [..] . The language should be like that as well, shouldn't it. You should know intuitively what everything does... what each function does and that. It might be a bit of a problem, but that's ... but ... that's what I think a good programming language should be like — that you know just by intuition ... without needing to read any manuals and that, ..

I. Do you think that's the case with BASIC?

A. BASIC is a bit like that I think. It's sort of, there are English words and suchlike, that tell you roughly what's happening ... I think. I haven't got anything to compare it with, I don't know any other languages.

Keith, who has worked in the computer industry for a few years and had programming as a hobby for even longer, also expresses this conception in the first interview, when describing his favourite language, Forth, in response to the same question:

K. Forth is a good language

I. What makes you say that?

K. Because you can do so much with it, as I said before. You can do everything yourself, define all the commands yourself.

I. Does it have any disadvantages?

K. It's hard to learn. Very hard to learn, it takes a long time.

I. What do you have to learn to be able to program in Forth?

K. You have to think in a new way. Say that you've been programming before in BASIC for example. There you've just got a line number, then you write a few commands, then take the next line number, write a few more commands. Then you can take a look, just at the program listing, and you see what the program does. You don't see that in Forth. What you've programmed, or made, are your own commands, so you don't really know what lies behind them. And then combining all these commands, that can be a bit tricky.

Here, and in the discussion that follows it, both Forth and BASIC are characterized by their internal structure, the sort of commands the languages contain and the way in which the programmer approaches them. Eric indicates the conception in this extract, but includes the sort of control a language has, the commands that enable the program to follow alternative paths through the commands:

I. Can you explain a bit more what you mean by those two things? That it is easy to use and easy to understand?

E. Ah ... that I should be able to understand, if I read a program then I should be able to understand what is in it .. and .. there should be simple commands that are easy to remember so that you can write a program, sort of thing. Then it's easy to use

I. And BASIC...?

E. That's quite ... well, it's easy to, easy to program in, because .. but then it's a bit harder to understand what you have written (laughs)

I. Why's that?

E. It is so ... unstructured. It's ... it jumps all over the place. When you are going through a program it jumps here and there so that it's hard to follow sort of. If it's a biggish program.

The code conception is actually only found in a few of the first interviews, but appears much more frequently in the sixth set, often in a more complex form, and it appears in the responses to several lines of questioning. Mark says:

I. And what do you see as the essential difference between those two languages? What is it that is so important?

M. Well, the difference I have noticed is certainly that, well, yes, the functional languages have maths as their basis so then you describe functions in a special way, or it's more like the basic mathematical functions, in as much as functions can be nested inside one another, or they are combined somehow. But in Pascal, there .. the structure is different, there are a mass of function statements that just, mm ...

and Greg says:

I. When it comes to programming, when you are actually programming, how does it feel? Are there any differences [in programming in ML and Pascal] there?

G. Of course there are. In Pascal you have to declare all your variables, but you don't have to do that in ML. And ... Well, there are big differences in all ways, because in ...in ML there is no what I think you call iteration, loops.

The second year student Frances says of her idea of a good programming language:

F. It should be able to .. it shouldn't be muddled, and it should be able to, it should be strict, there should be good strict rules so that it, it is sort of well organized. A program needs to have a section where it carries out all its calculations, different routines, subroutines, different places where it does its output. Of the languages we've studied so far I like ML best. I think I would say it's more logical, largely like maths, but there are logical forms that you can prove. In Pascal you make loads of declarations for variables and constants.

Programming language as a utility

This nomenclature also demands explanation. In computer use, the word utility is used for a program that is used in some way in connection with the system, rather than as an application a user might wish to use. Thus the system manager uses accounting utilities to keep track of who uses the computer and for how long, and housekeeping utilities for

keeping the storage media tidy. The word is used here to convey the sense that a programming language belongs with the computer system; it has large-scale features, such as efficiency or speed, and which can bring about effects in the computer. The large-scale features are often said to fit the language to one purpose more than another, but why this is so is left undetermined, often explained by the fact that the language was developed for that purpose. Another large-scale feature is that of speed, which might be explained by the fact that a language is compiled rather than interpreted, or that it is intended for a particular sort of computer.

Tim expresses this large-scale utility conception in the first interview, in the following extract, in which he refers to a long list of languages that the class had been given, and asked to state the extent to which they were familiar:

I. Why are there are so many programming languages? [..]

T. I really think there are unnecessarily many languages. There are, like Pascal here, that's a typical scientific language. If you want to control a robot for example, then it's better with Pascal. Cobol is a typical administrative language - you can set up big databases and keep customer records and so on. And then there are slightly different versions of all these. A lot of technology and finance go together, the administration of a small firm, so some of them might have been adapted to both bits, sort of thing. And then there's BASIC, that's basic, the start sort of, the beginner's language.

Neil, the most competent of the students at the start of the course, talks in similar vein in the first interview:

I. Can you point to anything that characterizes a good programming language? Can you make that distinction?

N. Yes you can. I mean, it depends on how you are going to use the language. If we take BASIC for example, then you've got to use it very simply. It's extremely simple to use. Multiplication tables and that sort of thing, it's very simple. But if you're going to do something more sophisticated, databases or such, ... for instance, you can't write a programming language in BASIC. Well, there are people who have done that, but I would say it's pretty hopeless, there's no point in doing it. But on the other hand, if you take the more advanced languages like Pascal and up, Modula2, Ada and all those, then to do a simple thing like a multiplication table, then it's a bigger job to get going than it is to use BASIC. So it has a lot to do with the level of the job. It depends on what you want to use the language for, you can't get away from that.

These two excerpts come from rather experienced programmers, both having already studied computer science at the undergraduate level. But in the first interview, it is generally this conception that is most commonly expressed, albeit in less articulate terms. In response to the question about the reasons for the existence of the large number of programming languages, Greg says:

G. Hmm, the reason for them must be that somebody has made them up, somebody thought they would be good.

I. Anything else?

G. Well, yes, different languages suit different things, don't they. Some are better for one thing, some for another. Some are better for word-processing and others are better for controlling measuring equipment and that sort of thing.

Similarly, Eric, having said that it is because different people like different languages, adds:

E. Then there are different areas of use. Some are meant just for maths and scientific applications.

I. How are they specially applicable to a particular area?

E. The commands and that, and instructions, they are made just for that area sort of.

Steve expresses the utility conception in the same way as a few others, namely that there are too many programming languages and some genius could combine them all:

S. (...) It would have been best if somebody could have coordinated it in some way, but it's not possible because of all the different machines and everything.

I. But is there anything good about having so many languages?

S. People earn money from it! Ah well, there are certainly some languages that can do one thing and others that can't cope at all with it, or they're more suitable for writing programs for one thing than another (... ...) But it would be good if there were some genius who could coordinate it so that we had some gigantic super language that could be used for more or less everything ...

In the sixth interview the same conception is still expressed, but with two difference. The first is that it now never the only conception expressed, and the second is that it is expressed in more complex terms. Thus John says:

J. A good program language, well you should be able to solve a lot of different kinds of problem without - it depends on the programmer himself as well of course - without the program getting too big and hard to keep a grip on ... and ... well sometimes it can be good ... there are different - sometimes it can be good if there are a lot of predefined functions in the language but sometimes it can be a disadvantage too, to keep track of them all

This starts by expressing the utility conception, but the last part indicates the code conception. Greg also shows these two conceptions in the same response:

G. It should be easy to learn, easy to program in, easy to understand for somebody from outside, who hasn't written the program but sort of comes along afterwards and maybe has to make changes to it, it should be easy for him. And it should be fast

I. In what way, fast?

G. Execution

I. When you have written the program, it should run fast?

G. Yes

Brian adds, bringing in the suitability for the computer:

B. And then it [a programming language] should also be such that it is easy for the machine to understand, if I can put it that way, so that it is easy to carry out and fast to execute.

The second year students often express this utility conception, referring especially to the demand that a programming language should enable programs to be easily read by others. Thus Holly says, having been asked what she sees as a good programming language:

H. Well, it depends. There are different aspects to it, it's actually difficult to answer that because if I write a program then I think that a good programming language, it should make it possible for somebody else to take over from me and that person should be able to understand what I have written.

Thus a programming language should have the large-scale property of creating easily understood programs.

Programming language as a means of communication

This conception of programming languages sees them as the link between the programmer and the computer, via the program that is written. An extension is that the user of the program is also linked in in some way.

The only clear expression of this conception in the first interview is from Mark, who has said that machine code is actually the perfect programming language, but it is hard to program in:

I. In what way is it perfect?

M. It is sort of perfect for the computer. It is the computer that sets the rules for what you can do basically. It is ... but then it's certainly difficult ... there isn't just one language, one machine code, that suits another computer. So they can't be used either...

The second year student Collette expresses this conception of a programming language, in that it should be able to communicate with the computer on the computer's terms, but extends it to a sort of communication with the problem to be solved in abstract terms, thus:

C. I think a good programming language is one in which you can go very close to the computer and make precise instructions so that you are almost at the machine level, but then at the same time move away and be very abstract.

Neil, having suggested several criteria for a good programming language, says:

N [...] And then, as I said before, it must have communication, a good system in which you can communicate with the computer, the user and so on.

This could be argued to express the utility conception, but the significant difference is that it refers explicitly to the links that exist between the user, the computer and the program, rather than the unspecified reasons for the large-scale features which the utility program has.

Programming language as a means of expression

The fourth conception that has been identified is one which sees the programming language as a means of expressing the solution which is devised for a problem. Some statements imply this directly, especially those in response to questions like, "what do you see as a good programming language?" Others, typically in discussion of programming in the two languages ML and Pascal, imply this conception less directly.

Here are three examples of the more direct statements. The only time this is said in the first set of interviews is when David says:

D. A good programming language should be able to do the things you want to do.

In the sixth set of interviews more imply this conception. For example, Robert says:

R. A good programming language, ... well, I suppose a programming language can't be best at everything. If you take a language that is specialized to one area maybe, like ML in maths, well, then you want that in that area you won't come across any limitations.

and Mark says:

M. It should be easy to learn to solve problems in it, that's what [a good programming language] should be.

The indirect statements implying this conception are often made when discussing the differences and similarities between programming in the two programming languages they have learned. Mark offers one example:

I. Are there any similarities between programming in ML and Pascal?

M. Oh sure. Absolutely. Because you, programming as programming, you have to solve the problem in some way, and you do that presumably in the same way for both, and then it's only a question of applying them in different ways so that you ... write the program itself. So the thinking that lies behind it is just the same, I think. But you have to approach it in a slightly different way when you write the program.

This conception is not necessarily related to the fact that Mark sees solving the problem as independent of writing the program in whichever language is called for. Tim expresses the opposite feeling, but still implies that the language is the means of expressing his solutions in the following extract:

I. What do you see as the essential differences in programming in ML and Pascal?

T. Mmm. In ML [..] there you have to prepare things much more carefully. You can't ... like in Pascal it's one program, you'll do this and this first, then that and that, and then the other. So then you can sort of think again, afterwards, when you are sitting and programming, then you can sort of, well, you might have forgotten a line somewhere, so you can just insert it. But in ML you sort of have to think backwards, I'm going to need this little function so I'll write it now, then I'll have to use that little function, so that needs writing. So you sort of have to break it up backwards so to speak.

Here the programming language is being used as part of the solving process, and determining the way in which this student approaches the problem.

Summary

Four conceptions of programming languages have been described here as evidenced by the first and sixth interviews with the students in the main study of learning programming, and interviews with the second year students. It would be wrong to imagine that a student consistently expresses only one of these conceptions throughout an interview. On the contrary, two or even three might be expressed, differentiated by the context. The important point at this stage of the study is that they do exist and are distinctly different.

Two of them have corresponding features in the theory of programming. The code conception corresponds to that part of programming which deals with the syntax and semantics of a programming language; syntax here refers to the words, the symbols that make up the language and semantics refers to the rules that determine the effect brought about by combinations of these words and symbols. The utility conception refers to the large-scale features, which roughly corresponds to the pragmatics of a language, or the way in which the language can be made to work for the programmer. The means of communication conception can be seen as referring to the aptness of the language for the computer in use, as well as its aptness in terms of input and output to the program user. The expressive medium conception refers to the aptness of the language to the solution to a problem, or even to the problem itself. While the first conception focuses into the language, the second looks at the language as an entity, and the other two focus on the relation perceived between the other parts of the programming system. This point will be taken up in the following section, on complexity in the conceptions of programming.

Discussion

Complexity in the conceptions of programming

Programming is a complex phenomenon, composed of only slightly less complex phenomena. The programming process can be seen as involving a problem of some sort, a computer which is to be used to help in reaching a solution to the problem, a programmer who draws on his skills to devise a program in some programming language, and the program's final use by a user and possible maintenance by other programmers. This is a picture which actually comes out of the interviews rather than being a theoretical treatment of the subject.

Three conceptions of programming were identified earlier in the paper, characterized by their principal orientation. They are not seen as forming a simple hierarchy - there is no best conception from the point of views of being a programmer, or learning to program. But there is a hierarchy of complexity, in that the computer orientation is inadequate for professional programmers; the problem orientation is adequate for some areas of work such as for researchers who want to solve their own problems with no thought for them being used again or by others; but the majority of professional programmers will need to have developed the third conception, the product oriented conception. But by no means is the third conception adequate on its own - even if that were feasible. In fact, as was said in the earlier section on these conceptions, this third conception subsumes the other two - the computer's role and the problem giving rise to programming are not missing from statements which are taken to indicate the product orientation, but they are in the background and focus is on the product.

A second factor in this complexity lies in the various factors involved in the programming process, and the conceptions held of them. Figure 1 shows these in a schematic form. This is no sense a cognitive map or mental model of programming, but an attempt to illustrate the factors involved in such a way that the conceptions of individuals can later be illustrated.

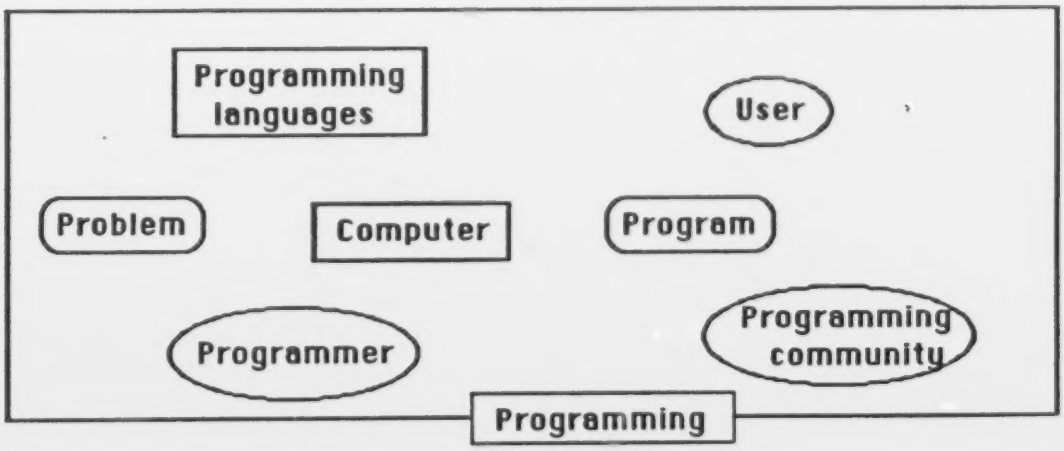

Figure 1. Schematic of the phenomena involved in the conception of programming

Figure 1 shows programming as being a system made up of problem, computer and product; programmer, user and programming community; and programming languages. The phenomena which have been analysed in this paper are seen in rectangular boxes, the

outer one representing programming in the global sense. With the help of such figures it should now be possible to take a broader look at what is said in individual interviews, or parts of interviews in different contexts. Arrows will be used to indicate the relationships that are expressed in the conception, with their direction showing the direction of the relationship if it is made clear, and their relative shading showing the relative emphasis of the relationship within the conception as a whole.

Learning to program

In previous phenomenographic studies, learning has been seen as the transition from one conception of the phenomenon in question to another, qualitatively better, conception. In the case of programming, it is proposed that a better way of seeing learning is as not only the development of a qualitatively more advanced conception of programming, but also to develop quantitatively more conceptions. By this is meant that a competent professional programmer, or software engineer, does not only need to acquire the product oriented conception of programming, but to acquire and subsume under it both the problem oriented and the computer oriented conceptions. Then handling these conceptions, being able to bring into play the best conception, or grouping, according to the needs of a particular context, is linked with efficiency of the programming skill.

A simple case to consider is that of Clive. In the first interview he says that programming is:

> C. It's to sit in front of the computer and .. ah .. hmm .. how can I put it ... to get the machine to achieve something, something that you want it to do.

Thus his conception of programming is computer oriented and very simple. It can be illustrated by Figure 2.

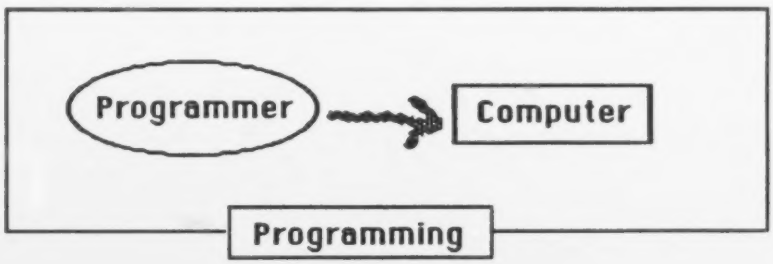

Figure 2. Clive's conception of programming, interview 1.

It is simply that the programmer makes the computer achieve something that is only specified in terms of the programmer's own wishes. The only phenomena present are the programmer (implicitly) and the computer (explicitly), and the only relationship is that from the programmer to the computer. While he refers to the product (something you want it to do), it is totally non-specific. In the sixth interview he says, still rather simply:

> C. It's to solve a task you get, to solve a problem. And ... to get it to work as well as possible.

which now takes in the problem and the product, in more specific terms. This can be illustrated by figure 3.

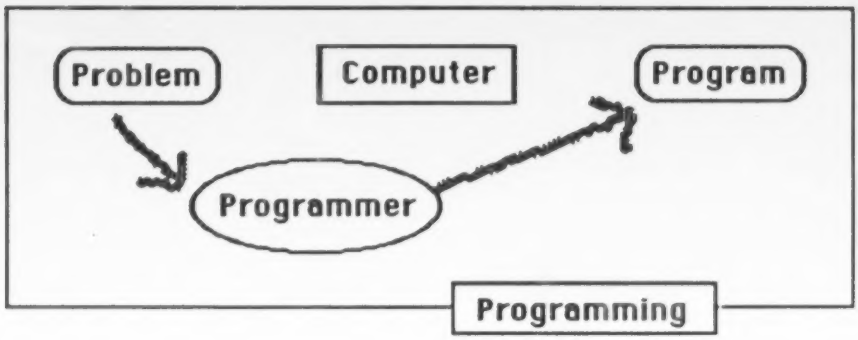

Figure 3. Clive's conception of programming, interview 6.

The problem is specified as something input to the programmer (a task you get), and you use the computer to make something, the product, that has the property of working as well as possible. The diagram shows that the programmer, the problem and the product are related with equal emphasis in a unidirectional string, while the computer is only implicitly involved. Clive can be said to have developed his conception of programming in the coarsest possible terms in that he has not only acquired a more advanced conception, but retains the less advanced ones as well.

Is that all that can be said about learning to program from the data analysed so far? This look at Clive's learning totally ignores the conceptions of the sub-phenomena. It also avoids considering the complexity of the conceptions, because Clive made such very simple statements and no more. Neil is quite a different case. In the first interview he says of programming:

I. What is programming, for you?

N. [...] Well, first of all it depends what you are going to do. But if you consider it in general, then you've got to sit down with the problem. I discovered that the first time I wrote a program that was going to be sold to somebody. I wrote the program and then I asked what it was going to be used for, and that was rather a mistake. [..] First you need to find out how to program before you start programming. And then it depends what language you are going to use [..] But as I said, first you've got to sort out the problem.

He continues in this vein, but here are included all the main aspects he takes up - sorting out the problem is most important, knowing how to program and mastering the programming language to be used. It can be illustrated by Figure 4.

This is clearly a problem-oriented conception of programming, where the relation to the problem is strongest, both as the problem affects the task of the programmer and as the programmer has to sort out the problem. The problem is presented to the programmer in some way, and the programmer has to tackle the problem before going further, as shown by the pair of arrows between the programmer and the problem in Figure 4. The program is only an incidental interest - it is not a "product" in the sense of the product-oriented conception. The computer is not specifically mentioned here, except that you need to know how to program, which at any level means using the computer. The programming language is given to the programmer.

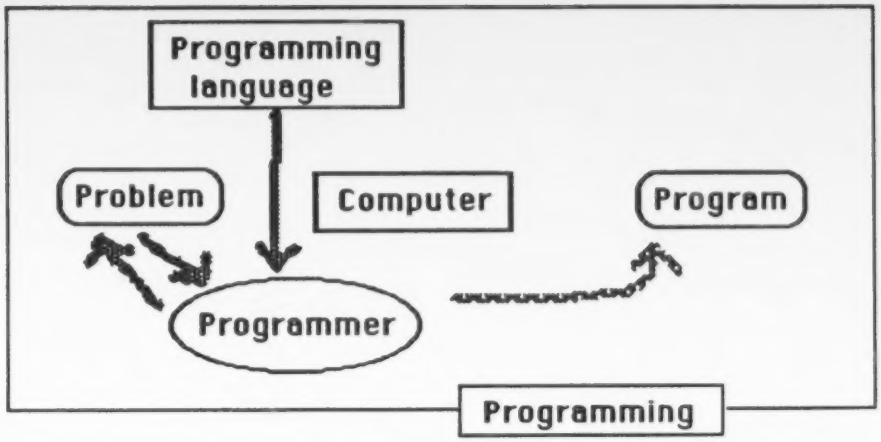

Figure 4. Neil's conception of programming, Interview 1.

In the sixth interview Neil says:

N. [...] It's to analyse a problem in a sensible way, to see a problem and then transf... or realise the problem in program code in whichever programming language you are using. And to do it in a structured and sensible way and preferable effectively, so that you use the least possible time to achieve the result. The result should be optimal on all fronts, whether it is program code, efficiency, exploiting computer power and so on. That's what you do yourself, but when you work with others you have to get into communication with your colleagues, first with the customer, understand what his problem is, not to build a castle when he wants a bungalow so to speak. And that's also a sort of aspect of programming, when you are are working with others, programming in a group, that you should effect a communication between them, that you have the same level of understanding of the problem. That's what I think programming is, but then the programming itself, the sitting and typing, that's only the result of all the thinking and analysis of the problem.

The significant difference between this and the previous quote is that this one extends the conception of programming to the product orientation, referring to both the user and the programmer's colleagues. It can be illustrated by Figure 5.

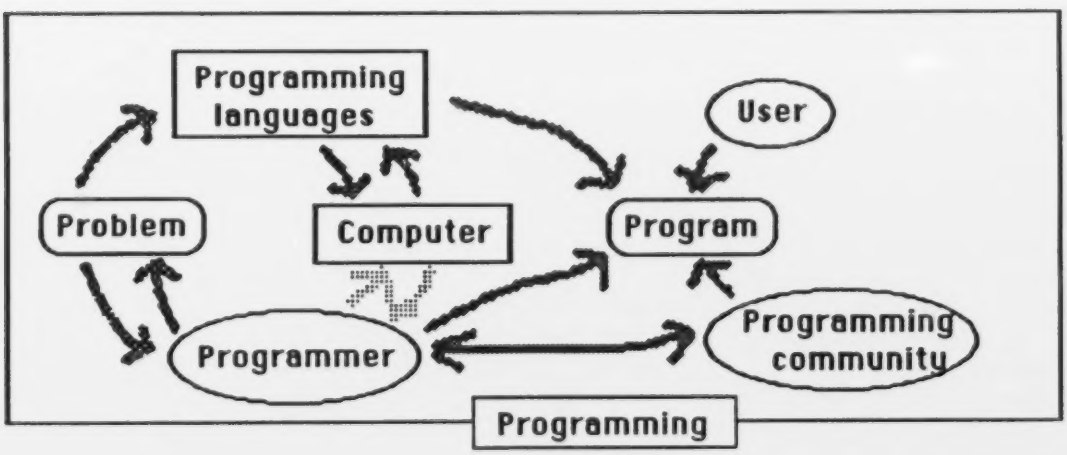

Figure 5. Neil's conception of programming, Interview 6.

This shows that the problem is still seen to be the most important input to the programming activity, but the programming community and the user's demands for the product are also

brought in, as is the quality of the product. In fact, all three conceptions of programming are separately indicated here, but altogether they go to make up the third, product oriented conception.

These figures could go one step further and attempt to show the conceptions indicated for the sub-phenomena. Neil's sixth interview, even the paragraph quoted, clearly shows the conception of programming languages as media for expressing a solution (realise the problem in program code in whichever programming language you are using) whereas in the first interview he indicated (not in the extract quoted) the utility conception (different languages are suited to different purposes). The fact that the arrow now goes between the programming language and the problem rather than between the language and the programmer as it did earlier illustrates this change. By systematically comparing the overall changes in the conception of programming and the changing conceptions in the sub-phenomena, it should be possible to build up a picture of learning to program in these terms.

The way ahead

The report has now described the current state of the study. It has come long enough to present these results, that appear to be quite stable across interviews held at different stages and with students at different levels of competence.

At the end of the previous section one plan was mentioned for developing our understanding of the overall picture of the activity of programming and learning to program. But all this is what was called at the beginning of this report "at the non-technical level". The whole report has been restricted to this level, the level of the large-scale phenomenon of programming. This is important and interesting in as much as it should be generally relevant, not maybe for all classes of people learning to program all languages, but certainly for more advanced school students and university-level students learning high-level languages for serious professional motives.

But this project is to study a particular group of students learning to program in two particular programming languages. So the second projected development is that of examining the material for its more technical aspects. The first of these is the problem-solving aspect, that comes in when the students were given unseen problems to solve, or asked to recount problems (or sometimes respond to short questions) that had been encountered in exercise classes, examinations and laboratory exercises, and is expected to contribute to the study of both technical and non-technical aspects of programming. The second is the aspect of learning constructs that are specific to the individual languages. This comes up not only in response to questions which explicitly aimed at uncovering such conceptions, but also in discussion of the characteristics of the languages, comparisons of them at the end of the course, and incidentally during the problem-solving sessions.

So this is the way ahead. Firstly, it is to trace learning of programming in the large, by examining changing conceptions of central phenomena and in particular the complexity of the conception of programming itself. Then it is to look at problem-solving and, thirdly, it is to examine conceptions of central constructs in ML and Pascal, and attempt to fit them into the overall picture.

References

Andersson, J. R., Farrell, R., & Sauers, R., (1984). Learning to program in LISP, *Cognitive Science, 8*, 87-129

Booth, S. A., (1987). Learning to program: A review of some recent investigations, *Publikationer från institution för pedagogik, Göteborgs Universitet.*

Johansson, B., F. Marton, L. Svensson, (1985). "An approach to describing learning as change between qualitatively different conceptions". In A. L. Pines and L. H. T. West (Eds.), *Cognitive Structure and Conceptual Change*, New York, Academic Press.

Lammers, S. (1987) *Programmers at Work*, New York: Microsoft press

Linn, M. C., (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher, 14*, 5-29

Marton, F., (1986). Phenomenography – A research approach to investigating different understandings of reality, *Journal of Thought, 21*, 28-49

Mayer, R. E., (1981). The psychology of how novices learn computer programming. *Computer Surveys, 1*, 121-141

Neuman, D., (1987). *The origin of arithmetic skills*, Göteborg: Acta Universitatis Gothoburgensis.

Renström, (1988), *Conceptions of Matter*, Göteborg: Acta Universitatis Gothoburgensis.

Soloway, E., (1985). From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research, 1*, 157-172.

Wikström, Å., (1986). Functional Programming using Standard ML. London, Prentice Hall

Appendix A

The programming languages involved in the study: Standard Meta-Language (SML or ML) and Pascal

Two languages are involved in this study of learning to program. The students who were followed and studied undertook a sixteen-week course of programming, in which the first half a language called Standard Meta-Language (SML) was used and the second half Pascal was mainly used. There is a didactic purpose in introducing programming in the two languages in this order, in that SML is thought to encourage the students to focus on the problem to be solved, as well as introducing a language with very powerful features, while Pascal is a more conventional language which the students will actually find more useful during the rest of their engineering studies. There follows a summary of some of the ideas behind the two languages which are relevant in the context of this paper. Further details are to be found in any text book on programming in the relevant languages, for example Wikström (1987) for SML.

Both SML (or ML as it is often called) and Pascal are *high-level* programming languages, SML being a *functional* language and Pascal an *imperative, procedural* language. The term high-level refers to a notional scale which extends from the computer itself to the problem which has to be solved via the computer. At various points on this scale different kinds of programming languages enable the programmer to specify and write an appropriate program. The specification that corresponds to the problem is in natural language (or something appropriately close to it) and the specification that corresponds to the computer implementation is written in terms of the electronic representations used by the particular computer. Machine code, a low-level language which refers specifically to the inner workings of the machine and data in specific locations in the machine, lies at one extreme. Progressing along the scale we meet assembler languages of codes which mirror the computer's own processes but have other useful functions added and allow flexible storage of data. Further yet are the high-level languages like Pascal, which have moved away from the computer's own instruction set and enable the programmer to use symbols and instructions which reflect the logic and processes of mathematics, and data structures which are more useful for handling numeric and textual data. At the far extreme would lie a natural language specification, if such were feasible, and SML exemplifies a step towards this in that it allows a mathematical representation of a problem to be specified in mathematical terms.

The two terms *functional* and *imperative* have a number of implications. First, a program written in SML consists of a number of expressions which both appear like, and actually are, mathematical functions. Correspondingly, the term imperative means that a program written in Pascal consists of a number of commands, instructing the computer how to proceed from an initial state to the state required. Pascal is also a *procedural* language, which refers to the fact that a program written in it describes procedures for the computer to execute. A second difference between the two kinds of language is that Pascal is one of the many programming languages that has been developed expressly for use on a particular design of computer, the von Neumann computer, which executes one instruction at a time, sequentially. The functional programming languages, on the other hand, are not designed for any particular machine, but are more generally designed on mathematical principles and can in principle be implemented on any computer.

A few words about functional programming in general are appropriate here. The family of functional programming languages, which also includes the much older and more widely spread LISP, all have their roots in the area of mathematics known as lambda calculus, developed by Alonzo Church in the 1930s in connection with his work on computability. (For a description of the development of functional languages see, for example, Wikström (1987)). Their characteristics are expressions which have the form of

functions, algorithms which are based on recursion, and data structures which are based on the list. The first two of these have their counterparts in mathematics as the function and induction while the elements of lists can be identified by the natural numbers. Thus to a large extent the programs written in ML are provable by induction over the natural numbers - their correctness can be confirmed by reason via their mathematical counterparts rather than being validated by experiment.

SML has a number of features which should be mentioned here, with reference to the features of Pascal. Firstly, two features of importance, and common to all strictly functional programming languages, is that of the environment of bindings and referential transparency. A functional program consists of a sequence of declarations and functional expressions, the values of which are computed when the program is executed. Each valuation causes a new set of data to be created, consisting of all the names used in the program and their current values. The pairs of names and values are called *bindings* and the set of bindings is called an *environment*. This is in contrast to the imperative or procedural languages which store the value of a variable in specific cells of memory and have to get round the problem of inadvertent overwriting of values by having local and global variables - local variables have a limited area of use (or scope) while global variables maintain their values in all areas of the program. Referential transparency refers to the fact that the only thing that matters about an expression in SML is the value it returns. Compare that with the expressions that occur in languages such as Pascal - there an expression might look like a function but bring about physical side-effects such as printing a value on paper, or storing a new datum in the computer's memory. SML's referential transparency means that an expression always returns the same value in a particular environment and never brings about a side-effect. This is of practical importance in programming for computers with parallel processors, for then a function must compute the same value irrespective of where it is executed.

A second feature of SML is that it is strongly *typed*, which means that every data value in a program is of a specific type, and functions can only operate on the type (or types) of data it has been defined for. Pascal is also a typed language, but not inherently so, whereas newer languages of the same family as Pascal (Ada for example) have similar typing features to those of SML. A *type* is a set of data objects of similar kind - all programmers are familiar with the integer, real and string types of the more common languages, and that one may not add an integer to a real number, much less add strings as though they were numbers. There is an extension in SML in the form of a *data type*, which is a type together with operations for working with data objects of the type. A data type can thus represent an object in the real world in program terms. A simple example might be a data type to represent a real world society membership. The objects to be represented are the members, each of which might be represented by a record with items for name, home address, work address, journal subscription address, membership status and society position. The operations necessary for maintaining the list are: making a new list, inserting a new member, deleting a former member, changing an item in a record specified by the member's name, and deleting an entire list. Each of these operations can be programmed as a function operating on a list of records containing the specified items, and the whole makes up a data type. Data types can be of two kinds, concrete or abstract. The example was of a *concrete data type* - the real objects are modelled by one particular representation (a list of records with specified items) which would be visible to the programmer using the data type. An *abstract data type* is made by abstracting away from the particular and visible representation, so that the programmer making use of it knows only of the results brought about by the operations without knowing specifically how the object is represented, and the representation could be changed without it affecting the data type's use.

SML has its roots in mathematics and its expressions closely resemble those of mathematics, but this does not mean that SML is only applicable to mathematical problems; however, the problem must be expressible in mathematical terms. Programs can be written in SML to perform all the usual programmable tasks, such as text processing, data base maintenance, and even such fundamental tasks as programming language development.

There are shortcomings in the version taught to the students studied in that its communication with the outside world is very limited, largely because of its strict referential transparency. It has no facilities for producing good output, either on the screen or on paper, and no graphics at all. It also has very limited aids for writing programs - none of the usual syntax checkers which are more or less standard in commercially used languages. Furthermore, the operating system which supports programs written in SML, while intrinsically powerful, is equally limited in terms of user-friendliness. Getting programs in SML into the computer and working can therefore be a tedious task, and running an SML program limits the user to getting straightforward numerical or textual results.

This summary of the features of SML and Pascal is very brief and rather lacking in detail; it also omits to describe general ideas connected with other programming languages. Its aim is only to cover some of the basic general ideas of the two languages — those the undergraduates in question might be expected to come across during their earliest studies. Further details will be taken up in future papers on conceptions of the more technical features learned.

Appendix B

The interviews in outline.

Of the total of six interviews with first year students of computer science and computer engineering, two are relevant for the analysis reported in this paper, namely the first and the sixth. Each interview contained five or six main topics, generally opened with a standard question and followed up with appropriate further questions. There follows an outline of these two first year interviews and that held with second year students.

Interview 1

1. Take up points from the questionnaire replies, for example their earlier experience with programming, at school and work.

2. Establish level of familiarity with the education they are about to embark on - Chalmers, the computer engineering program, the programming courses in general, and the programming languages SML and Pascal in particular.

3. Ask "What can computers be used for?" to open up the area of computer usage. Follow up with at least one question about future usage of computers.

4. Ask "Why do you think there are so many programming languages?" (referring to the questionnaire which had a list of languages with which they were asked to state their degree of familiarity). Follow up with at least "What do you think characterizes a good programming language?"

5. Ask "What is programming actually?", and follow up with "What do you think characterizes a good programmer?"

6. Ask "How do you think one learns to program?"

Interview 6

1. Ask "Now that you programming courses have finished, for this year at least, what do you see as the most important thing you have learned about programming?"

2. Start "I am going to repeat some questions I have asked before..." and continue "What is programming actually?"

3. Ask "What characterizes a good programming language?", probing further "And how do you see Pascal and SML in relation to this?"

4. Depending on the direction taken in the previous discussion, continue with a question to the effect "How can you compare SML and Pascal?", or "Can you point to any differences/similarities in the two languages/ using the two languages/ programming in the two languages?", in general probing their ideas of programming in general, and programming in SML and Pascal in particular.

5. Take up the concept of correctness of programs. Start "I too have taken a course on programming this term, and there the lecturer took up the idea of correctness. He said that now that programs had got so much bigger and more complex the big users of programs, like industrial companies and the US military for instance, had wearied of taking delivery of programs that had bugs in and had to be amended a lot. And so they are placing greater emphasis on getting programs that are correct." Then continue with three questions: "What is a correct program?", "How can you know that a program is correct?", and "How can you write programs that are correct from the outset?"

6. They had been given an essay to read, written by one of the teachers in programming the previous year, on learning and teaching programming, and on SML as an instructional language. They were asked for their general points of view on the essay, and asked what points they had found particularly interesting. Certain points were taken up specifically, if

they did not occur spontaneously. (This part of the interview has not been touched on in the current report.)

7. Ask "And finally, how do you think one learns to program?"

Second year interview

The interview opened with a general discussion of my aims for interviewing them, explaining that some of the questions might appear naive, because they were to be put to other students as well.

1. Ask "What can a computer be used for?", followed up by questions about how the computer carries out its tasks - how it does the things it can do - and about future computer usage.

2. Referring to the questionnaire they filled in when they were first year students, ask "Why are there so many programming languages?", "What characterizes a good programming language?", and then going on to ask specifically (if it had not been taken up spontaneously "What do you see as the main differences/similarities between SML and Pascal?"

3. Ask "What is programming, actually?", followed by "What characterizes a good programmer?"

4. Ask "Have your ideas of what programming is about been influenced by the course you have just completed?" and follow up as necessary.

5. Take up the idea of a correct program, as in first year Interview 6, question 5.

6. Present them with one of the harder questions from the exam paper:

> 5. A bag is a set in which elements can occur several times. Common operations on bags are *to create an empty bag, to insert an element in a bag, to determine how many times an element occurs in a bag, to form the union of two bags,* (the number of occurrences of elements is to be added), *to form the intersection of two bags* (the least number of occurrences of elements in the bags) and *to determine if two bags are equal.*
>
> a) Define an abstract data type in ML for bags with lists as representation. The operations union and intersection do not need to be implemented. The list should contain pairs of element and number of occurrences of the element.
>
> b) Define an abstract data type in ML for bags with functions from elements to whole numbers as representation. One of the six operations can not be implemented. Which, and why?
>
> c) Define an envelope in Pascal Plus to implement bags with the help of hash tables. Only the operations of inserting elements and finding the number of occurrences need be implemented. Open hashing is to be used, or in other words, elements which collide should be linked together in a list. Assume that the bag contains integers. (Hint: The function which counts the occurrences might be useful in the insert procedure.)

Ask, "Try to think back. What did you think when you read this problem in the exam?", "How did you start to solve it?" They were allowed to continue with the solution as far as they wished, but reminded rather often to try to recall how they behaved and thought during the exam. They were also asked about their general approaches to such problems.

7. Ask "How do you think one learns to program?"

REPORTS FROM DEPARTMENT OF EDUCATION AND EDUCATIONAL RESEARCH, GOTHENBURG UNIVERSITY

ISSN 0282-2156

To be ordered from Department of Education and Educational Research, Gothenburg University, P.O. Box 1010, S-431 26 MÖLNDAL, Sweden

---

Reuterberg, S-E. On comparing transition rate gains. 1985:01

Emanuelsson, I. and Svensson, A. Does the level of intelligence decrease? A comparison between thirteen-olds tested in 1961, 1966 and 1980. 1985:02

Lybeck, L. Research into science and mathematics education at Göteborg. 1985:03

Lybeck, L., Strömdahl, H. and Tullberg, A. Students' conceptions of amount of substance and its SI-unit 1 mol. A subject didactic study. 1985:04

Balke-Aurell, G. Testing testing methods. The Latin square design used in testing vocabulary by four methods. 1985:05

Sandström, B. Studies of the process of innovation in the comprehensive school. 1986:01

Lybeck, L. and Asplund Carlsson, M. Supervision of doctoral students. A case study. 1986:02

Oscarson, M. Native and non-native performance on a national test in English for Swedish students. A validation study. 1986:03

Ottosson, T. The world in mind: Mental models of spatial relations. 1987:01

Härnqvist, K. Background, education and work as predictors of adults skills. LING 8. 1989:01

Fransson, A. and Larsson, S. Who takes the second chance? Implementing educational equality in adult basic education in a Swedish context. 1989:02

Booth, S. Conceptions of Programming: A study into learning to program. 1990:01

ISSN 0282-2156

30 BEST COPY AVAILABLE